

AD-A126 086

THE STRUCTURE OF DIVIDE AND CONQUER ALGORITHMS(U) NAVAL 1/1
POSTGRADUATE SCHOOL MONTEREY CA D R SMITH 04 MAR 83
NP552-83-002

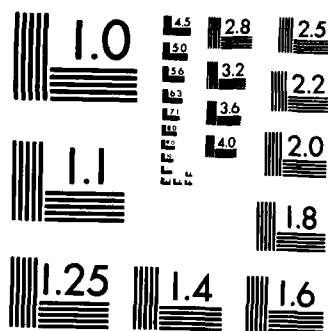
UNCLASSIFIED

F/G 12/1

NL

END

FILED
1
MAR 83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2

AD A 126086

NPS52-83-002

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THE STRUCTURE OF DIVIDE AND CONQUER ALGORITHMS

Douglas R. Smith

March 1983

DTIC
SELECTE
MAY 13 1983

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, Va 22217

08 28 001

DTIC FILE COPY

NAVAL POSTGRADUATE SCHOOL
Monterey, California

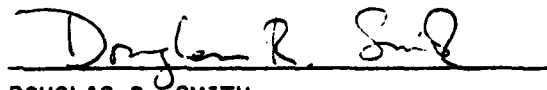
Rear Admiral J. J. Ekelund
Superintendent

D. A. Schradly
Acting Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Reproduction of all or part of this report is authorized.

This report was prepared by:



DOUGLAS R. SMITH
Assistant Professor
of Computer Science

Reviewed by:

Released by:



DAVID K. HSIAO, Chairman
Department of Computer Science



WILLIAM M. TOLLES
Dean of Research

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-83-002	2. GOVT ACCESSION NO. ADA126086	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Structure of Divide and Conquer Algorithms		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Douglas R. Smith		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N: RR000-01-100 N0001483WR30104
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE March 1983
		13. NUMBER OF PAGES 38
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Chief of Naval Research Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES Submitted for publication March 1983.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Algorithm design, divide and conquer, algebras, top-down programming, program schemes		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The structure of divide and conquer algorithms is represented by program schemes which provide a kind of normal-form for expressing these algorithms. A theorem relating the correctness of a divide and conquer algorithm to the correctness of its subalgorithms is given. Several strategies for designing divide and conquer algorithms arise from this theorem and we use them to formally derive algorithms for sorting a list of numbers, evaluating a propositional formula, and forming the cartesian product of two sets.		

The Structure of Divide and Conquer Algorithms¹

Douglas R. Smith
Department of Computer Science
Naval Postgraduate School
Monterey, California 93940
4 March 1983

ABSTRACT

The structure of divide and conquer algorithms is represented by program schemes which provide a kind of normal-form for expressing these algorithms. A theorem relating the correctness of a divide and conquer algorithm to the correctness of its subalgorithms is given. Several strategies for designing divide and conquer algorithms arise from this theorem and we use them to formally derive algorithms for sorting a list of numbers, evaluating a propositional formula, and forming the cartesian product of two sets.

0. Introduction

The advance of scientific knowledge often involves the grouping together of similar objects followed by the abstraction and representation of their common structural and functional features. Generic properties of the objects in the class are then studied by reasoning about this abstract characterization. The resulting theory may suggest strategies for designing objects in the class which have given characteristics. This paper reports on one such investigation into a class of related algorithms called "divide and conquer". We seek not only to gain a deeper and clearer understanding of the algorithms in this class, but to formulate this knowledge for the purposes of algorithm design. The essential structure of divide and conquer algorithms is expressed by a class of program schemes. We present a fundamental theorem relating the correctness of an instance of one of these schemes to the correctness of its parts. This theorem

¹ The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Avail and/or	
Special	
Dist	
Dist	
A	

provides a basis for designing divide and conquer algorithms in a formal way.

The principle underlying divide and conquer algorithms can be simply stated: if the problem posed by a given input is sufficiently simple we solve it directly, otherwise we decompose it into independent subproblems, solve the subproblems, then compose the resulting solutions. The process of decomposing the input problem and solving the subproblems gives rise to the term "divide and conquer" although "decompose, solve, and compose" would be more accurate.

We chose to explore the synthesis of divide and conquer algorithms for several reasons:

Structural Simplicity - Divide and conquer is perhaps the simplest program structuring technique which does not appear as an explicit control structure in current programming languages. Our description of the structure of divide and conquer algorithms is based on a view of them as computational homomorphisms between algebras on their input and output domains. Careful choice of programming language constructs allows us to express divide and conquer algorithms concisely and in accord with their essential structure as homomorphisms.

Computational Efficiency - Often algorithms of asymptotically optimal complexity arise from the application of the divide and conquer principle to a problem. Fast approximate algorithms for NP-hard problems frequently are based on the divide and conquer principle.

Diversity of Applications - Divide and conquer algorithms are common in programming, especially when processing structured data objects such as arrays, lists, and trees. Many examples of divide and conquer algorithms may be found in texts on algorithm design (e.g. [1,11]). Bentley [3] presents numerous applications of the divide and conquer principle to problems involving sets of objects in multidimensional space.

One of our goals is help formalize the process of designing algorithms to meet given specifications. Our approach in this paper is based on instantiating program schemes to obtain concrete programs satisfying a given specification. Related work on programming by instantiating program schemes is reported in [4,5,7,8,15]. Aside from the fact that we are concerned here with only one class of algorithms, our approach differs from these others mostly in focusing on formal techniques for deriving specifications for the uninterpreted operators in a program scheme.

In Section 1 we seek to acquaint the reader with some examples of divide and conquer algorithms. Algebraic notation introduced in Section 2 is used to present schemes in Section 3 characterizing the class of divide and conquer algorithms. The main result of this paper is a theorem showing how the correctness of a divide and conquer algorithm follows from its form and the correctness of its parts. In Section 4 we discuss the top-down design of divide and conquer algorithms and proceed with the derivation of a selection sort algorithm. In Section 5 we derive algorithms for a few more problems including the evaluation of Boolean expression and finding the cartesian product of two sets.

1. Examples of Divide and Conquer Algorithms

Applications of the divide and conquer principle are most naturally expressed by recursive programs. In Figure 1 we present a selection sort program expressed in an ad-hoc functional programming language (based on Backus' FP systems [2]) which we now summarize.

We use three data types: **IB** (Boolean values **TRUE** and **FALSE**), **IN** (natural numbers $0, 1, 2, \dots$), and **LIST(IN)** (linear lists of natural numbers e.g., **nil**, **(3)**, **(5, 2, 2, 7)**). Any element of these types is called an object, and if x_1, \dots, x_n for $n \geq 0$ are data objects then the n -tuple $\langle x_1, \dots, x_n \rangle$ is also a data object. The selector functions $_1, _2, \dots$ return the first, second, \dots elements of a tuple respectively. For example, $_1:\langle 3, 4 \rangle = 3$, $_2:\langle 3, 4 \rangle = 4$.

In a functional programming language programs are viewed as a hierarchy of functions. All functions map a data object to a data object. We use the notation $f:x$ to denote the result of applying the function (program) f to data object x . If a function requires n arguments for some $n > 1$, then it is applied to an n -tuple of objects. For the natural numbers we have the usual addition function, denoted $+$, and the comparison operators $<, \leq, =, \neq, \geq, >$. In deference to convention we allow infix notation for the arithmetic functions and relational operators, thus we equivalently write " $3+5$ " and " $+: \langle 3, 5 \rangle$ ". On the data type **LIST(IN)** we use the following functions: **Nil**, which returns the empty list (denoted **nil**); **List**, which maps a natural number into the list containing it; **First**, which returns the first element in a list; **Rest**, which returns its input list minus the first element; **Cons**, which adds a number to the front of a list (e.g. $\text{Cons}:\langle 2, (5, 4) \rangle = (2, 5, 4)$); **snoC**, (the inverse of **Cons**) which returns a 2-tuple containing the first element and the rest of the input list (e.g. $\text{snoC}:\langle 2, 5, 4 \rangle = \langle 2, (5, 4) \rangle$); and **Length**, which returns the length of a list. On all types we use **Id** as the identity function.

```

Ssort:x0 = if
    x0 = nil → x0 []
    x0 ≠ nil → Cons • (Id X Ssort) • Select:x0
fi

Select:x = if
    Rest:x = nil → snoC:x []
    Rest:x ≠ nil → Compose • (Id X Select) • snoC:x
fi

Compose:<v1,<v2,z>> = if
    v1 ≤ v2 → <v1,Cons:<v2,z>> []
    v1 ≥ v2 → <v2,Cons:<v1,z>>
fi

```

Figure 1: A Selection Sort Program

Functions are combined to yield new functions via the following combining forms. $f \cdot g$, called the composition of f and g , denotes the function resulting from applying f to the result of applying g to its argument.

For example: $\text{Length} \cdot \text{Rest}:(1,3,5) = \text{Length}:(\text{Rest}:(1,3,5))$
 $= \text{Length}:(3,5)$
 $= 2$

$f \times g$, called the product of f and g , is defined by

$f \times g:<x,y> = <f:x,g:y>.$

For example: $\text{Id} \times \text{Length}:<3,(1,3,5,7)> = <3,4>.$

If q_1, \dots, q_n are boolean functions or constants and f_1, \dots, f_n are functions or data objects then

$\text{if } q_1 \rightarrow f_1 \ [] \ \dots \ [] \ q_n \rightarrow f_n \ \text{fi}$

is a nondeterministic conditional form. During evaluation each of the boolean functions, called guards, are evaluated. If any of the guards are undefined, or

if none of the guards evaluate to TRUE, then the value of the form is undefined. Otherwise one of the guards, say q_i , which evaluates to TRUE is nondeterministically selected and the form evaluates to $f_i:x$. For example,

$$\text{if } \leq \rightarrow 1 \parallel \geq \rightarrow 2 \text{ fi}$$

is a simple if-fi form mapping $\mathbb{N} \times \mathbb{N}$ into \mathbb{N} and computing the minimum of two natural numbers. On application to $\langle 2,3 \rangle$ the guard " \leq " evaluates to TRUE thus the form evaluates to $1:\langle 2,3 \rangle = 2$. Note that on application to $\langle 3,3 \rangle$ both guards evaluate to TRUE thus either branch of the conditional can be taken. Although either branch can be taken the result is the same for this function.

We name functions by means of definitions. For example we can name the above if-fi form Min by means of the following definition

$$\text{Min} = \text{if } \leq \rightarrow 1 \parallel \geq \rightarrow 2 \text{ fi.}$$

For readability in definitions we allow the naming of arguments, replace selector function applications by the name of their result, and pretty print, so Min can be defined by

$$\begin{aligned} \text{Min:}\langle x,y \rangle = & \text{if} \\ & x \leq y \rightarrow x \parallel \\ & x \geq y \rightarrow y \\ & \text{fi.} \end{aligned}$$

The selection sort algorithm in Figure 1 works as follows. If the input is nil then nil is output. If the input is non-nil then a smallest element is split off and then prepended onto the result of recursively sorting the remainder of the input. The function Select evaluates as follows on the list $(2,5,1,4)$

$$\begin{aligned} \text{Select:}(2,5,1,4) &= \text{Compose} \cdot (\text{Id} \times \text{Select}) \cdot \text{snoC:}(2,5,1,4) \\ &= \text{Compose} \cdot (\text{Id} \times \text{Select}) : \langle 2, (5,1,4) \rangle \\ &= \text{Compose} : \langle 2, \langle 1, (5,4) \rangle \rangle \\ &= \langle 1, \text{Cons:} \langle 2, (5,4) \rangle \rangle \\ &= \langle 1, (2,5,4) \rangle \end{aligned}$$

where $\text{Select:}(5,1,4)$ evaluates to $\langle 1, (5,4) \rangle$ in a similar manner. Ssort when applied to $(2,5,1,4)$ evaluates as follows

$$\begin{aligned}
\text{Ssort:}(2,5,1,4) &= \text{Cons} \cdot (\text{Id} \times \text{Ssort}) \cdot \text{Select:}(2,5,1,4) \\
&= \text{Cons} \cdot (\text{Id} \times \text{Ssort}) : \langle 1, (2,5,4) \rangle \\
&= \text{Cons} : \langle 1, (2,4,5) \rangle \\
&= (1,2,4,5)
\end{aligned}$$

where $\text{Ssort:}(2,5,4)$ evaluates to $(2,4,5)$ in a similar manner.

Ssort and Select exemplify the structure of divide and conquer algorithms. In Ssort when the input is nil then the problem is solved directly, otherwise the input problem is decomposed via Select , the subproblems solved via the product $\text{Id} \times \text{Ssort}$, and the results composed by Cons . In Select when the input has length one then the problem is solved directly, otherwise the input is decomposed via snoC into a tuple of subinputs, the subinputs processed in parallel by $\text{Id} \times \text{Select}$, and the results composed by Compose . We call Select in Ssort and snoC in Select the decomposition operators. Cons in Ssort and Compose in Select are called composition operators. The identity function, Id , in both Ssort and Select is called an auxiliary operator.

Why introduce new language features here? We feel that the importance of divide and conquer algorithms is justification enough to require that a programming language allow their concise expression. We have introduced those linguistic features which allow divide and conquer programs to clearly reflect their essential structure. For example, the construction of decomposition operators is facilitated by allowing functions to return a tuple of objects. The product form allows us to directly express parallel processing of independent subproblems. In conditionals we are not forced to determine the order in which the guards are to be evaluated - they are conceptually evaluated in parallel. In addition, the language simplifies reasoning about and designing divide and conquer algorithms.

2. Algebraic Concepts

2.1 Program Termination

In designing divide and conquer algorithms we shall be concerned with ensuring that they terminate on all legal inputs. The usual method for showing the termination of a recursive program depends on the existence of a well-founded ordering on the input domain.

A structure $\langle W, \succ \rangle$ where W is a set and \succ is a binary relation on W is a well-founded set and \succ is a well-founded ordering on W if:

- 1) \succ is irreflexive: $u \not\succ u$ for all $u \in W$
- 2) \succ is asymmetric: if $u \succ v$ then $v \not\succ u$ for all $u, v \in W$
- 3) \succ is transitive: if $u \succ v$ and $v \succ w$ then $u \succ w$ for all $u, v, w \in W$
- 4) there is no infinite descending sequence $u_0 \succ u_1 \succ u_2 \succ \dots$ in W .

For example, \mathbb{N} (natural numbers) with the usual greater than relation $>$ forms the well-founded set $\langle \mathbb{N}, > \rangle$.

A recursive program P with input domain D can be shown to terminate on all inputs in the following way. First, a well-founded ordering \succ is constructed on D . Then, we show that for any $x \in D$ P applied to x only generates recursive applications (calls) to inputs x' for which $x \succ x'$. There can be no infinite sequence x_0, x_1, x_2, \dots such that applying P to x_i results in the application of P to x_{i+1} for $i \geq 0$ since the well-founded ordering does not allow $x_0 \succ x_1 \succ x_2 \succ \dots$.

Proposition 1. Let E be a set, let $\langle W, \succ_W \rangle$ be a well-founded set, and let $h: E \rightarrow W$ be a function from E into W . The relation \succ_E defined by:

$$u \succ_E u' \text{ iff } h(u) \succ_W h(u')$$

is a well-founded ordering on E .

Proof: 1) \succ_E is irreflexive - for any u , $h(u) \not\succ_W h(u)$, but then by definition $u \not\succ_E u$.

2) \succ_E is asymmetric - if $u \succ_E u'$ then $h(u) \succ_W h(u')$ and $h(u') \not\succ_W h(u)$ (by asymmetry of \succ_W) thus $u' \not\succ_E u$.

3) \succ_E is transitive - if $u \succ_E u'$ and $u' \succ_E u''$ then $h(u) \succ_W h(u')$ and $h(u') \succ_W h(u'')$. $h(u) \succ_W h(u'')$ follows by transitivity of \succ_W , then $u \succ_E u''$ follows by definition of \succ_E .

4) $\langle E, \succ_E \rangle$ has no infinite decreasing sequence - if $u_0 \succ_E u_1 \succ_E u_2 \succ_E \dots$ then $h(u_0) \succ_W h(u_1) \succ_W h(u_2) \succ \dots$ contradicting the well-foundedness of $\langle W, \succ_W \rangle$. QED

Proposition 1 enables us to establish a well-founded ordering on $\text{LIST}(\mathbb{N})$ (list of natural numbers) by simply finding a function from $\text{LIST}(\mathbb{N})$ to \mathbb{N} . A suitable primitive function is Length , so we may define

$$x \succ y \text{ iff } \text{Length}:x > \text{Length}:y$$

for all $x, y \in \text{LIST}(\mathbb{N})$. By Proposition 1 we conclude that $\langle \text{LIST}(\mathbb{N}), \mathcal{F} \rangle$ is a well-founded set.

2.2 Many-Sorted Algebras

Algebraic concepts are playing an increasingly important role in formulating the fundamental notions of computer science. In this paper we show that divide and conquer algorithms can be usefully characterized algebraically as homomorphisms between appropriately defined algebras on the input and output domains. In this section we present the basic terminology of many-sorted algebras based on and extending the notation of ADJ [9,10].

For any $n \in \mathbb{N}$ let $\underline{n} = \{1, 2, \dots, n\}$. As usual the cartesian product of A_1, A_2, \dots, A_n is written $A_1 \times A_2 \times \dots \times A_n$ and denotes $\{ \langle a_1, a_2, \dots, a_n \rangle \mid a_i \in A_i \text{ for } i \in \underline{n} \}$. Parentheses are used for nesting so

$$A_1 \times (A_2 \times A_3) = \{ \langle a_1, \langle a_2, a_3 \rangle \rangle \mid a_1 \in A_1, a_2 \in A_2, a_3 \in A_3 \}$$

the set of 2-tuples whose first component belongs to A_1 , and whose second component belongs to $A_2 \times A_3$.

Generally, we use the term many-sorted algebra to denote a collection of sets equipped with operators defined on cartesian products of the sets. Let S denote a nonempty set of symbols called sorts and $\hat{s} \in S$ be a distinguished sort called the principal sort. A finite \hat{s} -oriented S -sorted signature Σ is a finite set of operator symbols $\{\sigma_1, \dots, \sigma_r\}$, $r \geq 1$, where for $1 \leq i \leq r$, σ_i has type $\langle w_i, \hat{s} \rangle$ where $w_i \in S^*$ and $w_i = w_{i1} \dots w_{in_i}$, $n_i \geq 0$. Let $\langle A_s \rangle_{s \in S}$ be an S -indexed family of sets. If $w \in S^*$ and $w = w_1 w_2 \dots w_n$ then A^w denotes the cartesian product $A_{w_1} \times A_{w_2} \times \dots \times A_{w_n}$. Letting λ denote the empty string, A^λ denotes the set consisting of the 0-tuple, $\{\langle \rangle\}$. A Σ -algebra A consists of a family of sets $\langle A_s \rangle_{s \in S}$ called the carriers of A , and a set of operators denoted $\sigma_i|_A$ $i=1, \dots, r$, where $\sigma_i|_A: A^{w_i} \rightarrow A_{\hat{s}}$. $A_{\hat{s}}$ will be called the principal carrier of A . A Σ -algebra A will be written $A = \{ \langle C_1, \dots, C_k \rangle, \{f_1, \dots, f_r\} \}$ where C_1, \dots, C_k are the carriers of A and f_1, \dots, f_r are its operators. A Σ -algebra will be called a composition algebra.

We shall be interested in composition algebras which 1) allow each element of the principal carrier to be expressed as a composition of other elements, and 2) compose smaller elements into larger elements. For example, on the domain $\text{LIST}(\mathbb{N})$ consider the operators

$$\text{Nil}: \rightarrow \text{LIST}(\mathbb{N}) \quad (\text{e.g., Nil}: \langle \rangle = \text{nil})$$

$List:N \rightarrow LIST(N)$ (e.g., $List:3 = (3)$)

$Cons:N \times LIST(N) \rightarrow LIST(N)$ (e.g., $Cons:<3,(1,4)> = (3,1,4)$).

Every list of natural numbers can be expressed as either a composition by $Cons$ ($Cons:<i,y>$ for some $i \in N$ and $y \in LIST(N)$) or by Nil , thus

$$\langle \{LIST(N), Nil\}, \{Cons, Nil\} \rangle$$

is a composition algebra for $LIST(N)$. For the domain $LIST(N)-nil$, the operators $Cons$ and $List$ allow expression of each non-nil list as a composition by $Cons$ ($Cons:<i,y>$ for some $i \in N$ and $y \in LIST(N)-nil$) or by $List$ ($List:i$ for some $i \in N$), thus

$$\langle \{LIST(N)-nil, Nil\}, \{Cons, List\} \rangle$$

is a composition algebra for $LIST(N)-nil$.

Let A and B be Σ -algebras and let $H = \langle h_s \rangle_{s \in S}$ be an S -indexed family of functions where for each $s \in S$, $h_s: A_s \rightarrow B_s$. If $w = w_1 w_2 \dots w_n$ let h^w denote the product function $h_{w_1} \times h_{w_2} \times \dots \times h_{w_n}$. Thus if $a \in A^w$ then

$$h^w:a = \langle h_{w_1}:a_1, h_{w_2}:a_2, \dots, h_{w_n}:a_n \rangle.$$

h^{λ} denotes the unique function mapping A^{λ} to B^{λ} , also written $Id_{\langle \rangle}$. $H = \langle h_s \rangle_{s \in S}$ is a $(\Sigma-)$ homomorphism from A to B if for each operator symbol σ_i and $a \in A^{w_i}$

$$h_s \cdot \sigma_i A : a = \sigma_i B \cdot h^{w_i} : a.$$

i.e. the diagram in Figure 2 commutes.

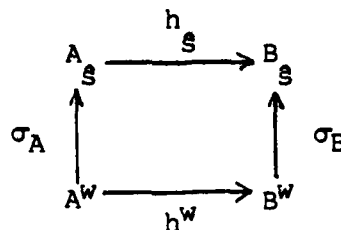


Figure 2: Commutative Diagram of a Σ -homomorphism.

A Σ^{-1} -algebra A is a family of sets $\langle A_s \rangle_{s \in S}$ and operators $\sigma_A: A_s \rightarrow A^w$ for each $1 \leq i \leq r$. A Σ^{-1} -algebra will be called a decomposition algebra. We shall be interested in decomposition algebras which 1) allow each element of the principal carrier to be decomposed into other elements, and 2) decompose larger elements into smaller elements. For example, on the domain $LIST(N)$ we can define operators which are the inverses of the composition operators considered above.

$$liN:LIST(N) \rightarrow \quad (e.g. liN:nil = \diamond)$$

$$tsiL:LIST(N) \rightarrow N \quad (e.g. tsiL:(3) = 3)$$

$$snoC:LIST(N) \rightarrow N \times LIST(N) \quad (e.g. snoC:(3,1,4) = \langle 3, (1,4) \rangle)$$

Every list of natural numbers can be decomposed either by $snoC$ or liN , thus

$$\langle \{LIST(N), N\}, \{snoC, liN\} \rangle$$

is a decomposition algebra for $LIST(N)$. For the domain $LIST(N)-nil$, the operators $snoC$ and $tsiL$ allow the decomposition of each non-nil list into non-nil lists and natural numbers, thus

$$\langle \{LIST(N)-nil, N\}, \{snoC, tsiL\} \rangle$$

is a decomposition algebra for $LIST(N)$.

Let A be a Σ^{-1} -algebra, B a Σ -algebra, and let $H = \langle h_s \rangle_{s \in S}$ be an S -indexed family of functions such that for each $s \in S$ $h_s: A_s \rightarrow B_s$. H is a $(\Sigma^{-1}\Sigma)$ -homomorphism from A to B if for each $x \in A_s$ such that $\sigma_A: x$ is defined

$$h_s: x = \sigma_B \cdot h^w \cdot \sigma_A: x \quad (2.1)$$

i.e., the diagram in Figure 3 commutes. For example, let $S = \{c, \delta\}$ and let

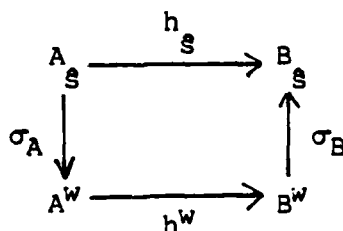


Figure 3: Commutative Diagram of a $\Sigma^{-1}\Sigma$ -homomorphism.

$\Sigma = \{\sigma_1, \sigma_2\}$ be a S -sorted signature where σ_1 has type $\langle \lambda, s \rangle$ and σ_2 has type $\langle cs, s \rangle$. Consider LS and LC which are Σ^{-1} and Σ -algebras respectively where:

$$LS = \langle \{N, LIST(N)\}, \{liN, Select\} \rangle$$

$$LC = \langle \{N, LIST(N)\}, \{Nil, Cons\} \rangle.$$

LS has carriers $LS_c = N$ and $LS_s = LIST(N)$ and operators

$$Select: LIST(N) \rightarrow N \times LIST(N) \text{ and}$$

$$liN: LIST(N) \rightarrow \{\langle \rangle\}.$$

$Select$ splits a list of natural numbers into its least element and the rest of the list as discussed earlier. LC has carriers $LC_c = N$ and $LC_s = LIST(N)$ and operators

$$Cons: N \times LIST(N) \rightarrow LIST(N) \text{ and}$$

$$Nil: \{\langle \rangle\} \rightarrow LIST(N).$$

Letting h_s be the function $Sort$, which sorts a list of numbers, and h_c the identity function Id , we have a natural homomorphism from LS to LC . First, $Sort$ and Id have the required domains and codomains:

$$Id: N \rightarrow N \quad (h_c: LS_c \rightarrow LC_c)$$

$$Sort: LIST(N) \rightarrow LIST(N) \quad (h_s: LS_s \rightarrow LC_s)$$

and the homomorphism condition (2.1) is satisfied: for any $x \in LIST(N)$ such that $liN:x$ is defined (i.e. $x = nil$)

$$Sort:x = Nil \cdot Id_{\langle \rangle} \cdot liN:x \quad (h_s:x = \sigma_{1LC} \cdot h^{\lambda}_{\lambda} \cdot \sigma_{1LS}:x)$$

and for any $x \in LIST(N)$ such that $Select:x$ is defined (i.e. $x \neq nil$)

$$Sort:x = Cons \cdot (Id \times Sort) \cdot Select:x. \quad (h_s:x = \sigma_{2LC} \cdot h^{cs}_{cs} \cdot \sigma_{2LS}:x)$$

This homomorphism, of course, is the essence of a selection sort algorithm. When the input x is nil we can sort directly, otherwise we decompose x into a number i and a list y , sort y , then $Cons$ i onto the result.

3. Divide and Conquer Algorithms: Form and Function

In this section we present notation expressing the form (via program schemes) and function (via specifications) of divide and conquer algorithms. We also present a fundamental theorem showing how the functionality of a divide and conquer program follows from its form and the functionalities of its parts. First we consider the expression of functionality.

3.1 Specifications

Specifications are a precise notation for describing the problem (or function) we desire to solve without necessarily indicating how to solve (or compute) it. For example, the problem of decomposing a list of natural numbers into its smallest element and the remainder of the list may be specified as follows.

$$\text{Select: } x = \langle i, z \rangle \text{ such that } x \neq \text{nil} \Rightarrow i \leq \text{Bag: } z \wedge \text{Bag: } x = \text{Add: } \langle i, \text{Bag: } z \rangle$$
$$\text{where Select: LIST(IN) } \rightarrow \text{ IN } \times \text{LIST(IN)}.$$

The problem is named Select which is a function from lists of natural numbers to 2-tuples consisting of a natural number and a list. Naming the input x and the output $\langle i, z \rangle$, the formula " $x \neq \text{nil}$ ", called the input condition, expresses any restrictions on the inputs we can expect to the problem. The formula " $i \leq \text{Bag: } z \wedge \text{Bag: } x = \text{Add: } \langle i, \text{Bag: } z \rangle$ ", called the output condition, expresses the conditions under which $\langle i, z \rangle$ is an acceptable output with respect to input x . The function Bag maps a list into the bag (multiset) of elements contained in it (e.g. $\text{Bag: } (1, 5, 2, 2) = \{1, 5, 2, 2\} = \text{Bag: } (1, 2, 5, 2)$). $i \leq \text{Bag: } z$ asserts that each element in the list z is no less than i . The function $\text{Add: } \langle i, b \rangle$ returns the bag containing i in addition to all elements of bag b . $\text{Bag: } x = \text{Add: } \langle i, \text{Bag: } z \rangle$, asserts that the multiset (bag) of elements in the input list x is the same as the multiset of elements in z with i added.

Generally, a specification Π has the form

$$\Pi: x = z \text{ such that } I: x \Rightarrow O: \langle x, z \rangle$$
$$\text{where } \Pi: D \rightarrow R.$$

We ambiguously use the symbol Π to denote both the problem, its specification, and a solution to the problem. Here the input and output domains are D and R respectively. The input condition I expresses any properties we can expect of inputs to the desired program. Inputs satisfying the input condition will be called legal inputs. If an input does not satisfy the input condition then we

don't care what output, if any, the program produces. The output condition O expresses the properties that an output object should satisfy. Any output object z such that $O:\langle x, z \rangle$ holds will be called a feasible output with respect to input x . More formally, a specification Π is a 4-tuple $\langle D, R, I, O \rangle$ where

D is a set called the input domain,

R is a set called the output domain,

I is a relation on D called the input condition, and

O is a relation on $D \times R$ called the output condition.

Program F satisfies specification $\Pi = \langle D, R, I, O \rangle$ if

$$\forall x \in D [I:x \Rightarrow O:\langle x, F:x \rangle]$$

is valid in a suitable first-order theory, i.e., if on each legal input F computes a feasible output.

Let S be a set of sorts with principal sort \hat{s} . $\hat{\Pi} = \langle E, T, J, P \rangle$ denotes an S -sorted family of problems where E and T are S -sorted families of sets, for each $s \in S$ J_s is a relation on E_s and P_s is a relation on $E_s \times T_s$. For each $s \in S$ let $\hat{\Pi}_s$, called a component problem, denote the problem specification $\langle E_s, T_s, J_s, P_s \rangle$. $\hat{\Pi}_{\hat{s}}$ will be called the principal problem and for each $s \in S - \hat{s}$ $\hat{\Pi}_s$ will be called an auxiliary problem.

3.2 The Form of Divide and Conquer Algorithms

Let S be a sort set with principal sort \hat{s} and let Σ be a finite \hat{s} -oriented S -sorted signature where $\Sigma = \{\sigma_1, \dots, \sigma_r\}$, $r \geq 1$, and for $1 \leq i \leq r$, σ_i has type $\langle w_i, \hat{s} \rangle$ where $w_i \in S^*$ and $w_i = w_{i1} \dots w_{in_i}$, $n_i \geq 0$. A Σ -divide and conquer algorithm has the form

$$\begin{aligned} f_{\hat{s}}:x &= \text{if} \\ &\quad q_1:x \rightarrow \sigma_{1T} \cdot f^{w_1} \cdot \sigma_{1E}:x[] \\ &\quad \dots \\ &\quad q_r:x \rightarrow \sigma_{rT} \cdot f^{w_r} \cdot \sigma_{rE}:x \\ &\text{fi.} \end{aligned}$$

where

1. E is a Σ^{-1} -algebra
2. T is a Σ -algebra
3. $F = \langle f_s \rangle_{s \in S}$ is an S -indexed family of functions where $f_s: E_s \rightarrow T_s$

4. q_i for $i \in \underline{r}$, is a predicate on E_s .

The operators in E and T are called the decomposition and composition operators respectively. Each f_s for $s \in S - \hat{s}$ is called an auxiliary function and $f_{\hat{s}}$ is called the principal function. In these terms the program's behavior can be described as follows: Given input x , a guard q_i which evaluates to TRUE is selected nondeterministically. Input x is decomposed by the decomposition operator σ_{i_E} into a tuple of subinputs. This tuple is then processed in parallel by the function product f^{wi} and the results composed by the composition operator σ_{i_T} . In order for the algorithm to terminate not all the branches of the conditional can contain recursive calls. The nonrecursive branches treat with those inputs which can be solved directly.

If we view the guards q_i for $i \in \underline{r}$ as characterizing the set of inputs on which the corresponding decomposition operator σ_{i_E} is defined, then the divide and conquer algorithm clearly expresses F as a homomorphism from the decomposition algebra E to the composition algebra T .

3.3 Correctness of a Divide and Conquer Algorithm

The main theoretical result of our paper is the following theorem which shows how the correctness of the whole divide and conquer algorithm follows from the correctness of its parts. Conditions (1), (2), and (3) of Theorem 1 simply provide the form of a specification for the parts of a Σ -divide and conquer algorithm. The most interesting condition is the "separability" condition (4). It is the principal link between the functionality of the algebras E and T , the auxiliary problems $\hat{\Pi}_s$, and the given principal problem. In words it states that if input x_0 decomposes into subinputs x_1, \dots, x_n , and z_1, \dots, z_n are feasible outputs with respect to these subinputs respectively, and z_1, \dots, z_n compose to form z_0 then z_0 is a feasible solution to input x_0 . Loosely put: feasible outputs compose to form feasible outputs. Condition (5) asserts that for each legal input at least one of the guards holds.

Theorem 1: Let S be a set of sorts with principal sort \hat{s} and let Σ be a finite \hat{s} -oriented S -sorted signature. Let E be a Σ^{-1} -algebra, T be a Σ -algebra, $\hat{\Pi}$ a S -sorted family of specifications, F a S -sorted family of functions where for each $s \in S$ $f_s: E_s \rightarrow T_s$. Let \succ be a well-founded ordering on $E_{\hat{s}}$ and for each $i \in \underline{r}$ let O_{i_E} and O_{i_T} be relations on E^{swi} and T^{swi} respectively. If

- (1) (Specification of σ_E) the decomposition operator σ_{iE} , for $i=1, \dots, r$, satisfies the specification

$$\sigma_{iE}:x_0 = \langle x_1, \dots, x_{n_i} \rangle \text{ such that } q_i:x_0 \wedge J_{\underline{s}}:x_0 \Rightarrow \bigwedge_{j \in \underline{n_i}} (J_{wi_j}:x_j \wedge (wi_j = \underline{s} \Rightarrow x_0 \neq x_j)) \wedge Oi_E:\langle x_0, x_1, \dots, x_{n_i} \rangle$$

where $\sigma_E:E_{\underline{s}} \rightarrow E^{wi}$

- (2) (Specification of σ_T) the composition operator σ_{iT} , for $i=1, \dots, r$, satisfies the specification

$$\sigma_{iT}:\langle z_1, \dots, z_{n_i} \rangle = z_0 \text{ such that } Oi_T:\langle z_0, z_1, \dots, z_{n_i} \rangle$$

where $\sigma_T:T^{wi} \rightarrow T_{\underline{s}}$

- (3) (Solutions to Auxiliary Problems) for each $s \in S - \underline{s}$ f_s satisfies specification

$$\hat{\Pi}_s:x = z \text{ such that } J_s:x \Rightarrow P_s:\langle x, z \rangle$$

where $\hat{\Pi}_s:E_s \rightarrow T_s$

- (4) (Separability of P) the following formula is valid for each $i \in \underline{r}$:

$$\forall \langle x_0, x_1, \dots, x_{n_i} \rangle \in E^{swi} \forall \langle z_0, z_1, \dots, z_{n_i} \rangle \in T^{swi}$$

$$[Oi_E:\langle x_0, x_1, \dots, x_{n_i} \rangle \wedge \bigwedge_{j \in \underline{n_i}} P_{wi_j}:\langle x_j, z_j \rangle \wedge Oi_T:\langle z_0, z_1, \dots, z_{n_i} \rangle \Rightarrow P_{\underline{s}}:\langle x_0, z_0 \rangle]$$

- (5) (Definition of the guards) For all $x \in E_{\underline{s}}$ $J_{\underline{s}}:x \Rightarrow \bigvee_{j \in \underline{r}} q_j:x$

then the divide and conquer program

$$f_{\underline{s}}:x = \text{if}$$

$$q_1:x \rightarrow \sigma_{1T} \cdot f^{wi}_1 \cdot \sigma_{1E}:x \quad []$$

$$\dots$$

$$q_r:x \rightarrow \sigma_{rT} \cdot f^{wr}_r \cdot \sigma_{rE}:x$$

fi

satisfies specification $\hat{\Pi}_{\underline{s}} = \langle E_{\underline{s}}, T_{\underline{s}}, J_{\underline{s}}, P_{\underline{s}} \rangle$.

Proof: To show that $f_{\underline{s}}$ satisfies $\hat{\Pi}_{\underline{s}} = \langle E_{\underline{s}}, T_{\underline{s}}, J_{\underline{s}}, P_{\underline{s}} \rangle$ we will prove

by structural induction² on E_s .

Let x be an arbitrary object in E_s such that $J_s:x$ holds and assume (inductively) that $J_s:y \Rightarrow P_s:\langle y, f_s:y \rangle$ holds for any $y \in E_s$ such that $x \rightarrow y$. From $J_s:x$ and condition (5) it follows that $q_i:x$ holds for some $i \in \underline{r}$. By the semantics of the if-fi construct $f_s:x$ can evaluate to $\sigma_{i_T} \cdot f^{wi} \cdot \sigma_{i_E}:x$. We will show that $P_s:\langle x, f_s:x \rangle$ by using the inductive assumption and modus ponens on the separability condition. Since $q_i:x \wedge J_s:x$ holds and σ_{i_E} satisfies its specification in condition (1), the output condition of σ_E also holds. Let $\sigma_{i_E}:x = \langle x_1, \dots, x_{n_i} \rangle$. We have for each $j \in \underline{n_i}$ $J_{wi_j}:x_j$. Consider x_j for each $j \in \underline{n_i}$. If $wi_j \neq s$ then by condition (3)

$$J_{wi_j}:x_j \Rightarrow P_{wi_j}:\langle x_j, f_{wi_j}:x_j \rangle$$

and we infer by modus ponens $P_{wi_j}:\langle x_j, f_{wi_j}:x_j \rangle$. If on the other hand $wi_j = s$ then by condition (1) we have $x_0 \rightarrow x_j$ and thus by our inductive assumption

$$J_{wi_j}:x_j \Rightarrow P_{wi_j}:\langle x_j, f_{wi_j}:x_j \rangle.$$

Again we infer $P_{wi_j}:\langle x_j, f_{wi_j}:x_j \rangle$ by modus ponens. By condition (2) we have

$$\sigma_{i_T}:\langle \sigma_{i_T}:\langle f_{wi_1}:x_1, \dots, f_{wi_n}:x_n \rangle, f_{wi_1}, \dots, f_{wi_n} \rangle$$

where

$$\sigma_{i_T}:\langle f_{wi_1}:x_1, \dots, f_{wi_n}:x_n \rangle = f_s:x.$$

We have now established the antecedent of condition (4) enabling us to infer $P_s:\langle x, f_s:x \rangle$. QED

Notice that in Theorem 1 the form of the subalgorithms σ_{i_E} , σ_{i_T} , and f_s for $s \in S - \hat{s}$ is not relevant. All that matters is that they satisfy their respective specifications. In other words, their function and not their form matters with respect to the correctness of the whole divide and conquer algorithm.

² Structural induction on a well-founded set $\langle W, \rightarrow \rangle$ is a form of mathematical induction described by

$$\forall x \in W \forall y \in W [x \rightarrow y \wedge Q:y \Rightarrow Q:x] \Rightarrow \forall x \in W Q:x$$

i.e., if $Q:x$ can be shown to follow from the assumption that $Q:y$ holds for each y such that $x \rightarrow y$, then we can conclude that $Q:x$ holds for all x .

4. The Design of Divide and Conquer Algorithms

4.1 A Problem Reduction Approach to Design

Design is a goal-directed activity and this is the primary reason for the importance of top-down design methods. One form of top-down design, which we call problem reduction, may be described by a two phase process - the top-down decomposition of problem specifications and the bottom-up composition of programs. In practice these phases are interleaved but it helps to understand them separately. Initially we are given a specification Π . In the first phase we create an overall program structure for Π , which fixes certain gross features of the desired program. Some parts of the structure are at first underdetermined but their functional specifications are worked out so that they can be treated as relatively independent subproblems to be solved at a later stage. Next we work in turn on each of the subproblem specifications, and so on. This process of creating program structure and decomposing problem specifications terminates in primitive problem specifications which can be solved directly, without reduction to subproblems. The result is a tree of specifications with the initial specification at the root and primitive problem specifications at the leaves. The children of a node represent the subproblem specifications written (or derived) as we create program structure.

The second phase involves the bottom-up composition of programs. Initially each primitive problem specification is solved to obtain a program (which is often a programming language operator). Subsequently whenever each of the subproblem specifications generated when working on specification Π have solutions, these subproblem solutions are assembled into a program for Π .

We advocate [13,14] a formal counterpart to the problem reduction approach based on the use of program schemes. A scheme provides a standard overall structure for the desired program and its uninterpreted operator symbols stand for the underdetermined parts of the structure. To use a scheme we require a corresponding design strategy. Given a problem specification Π a design strategy derives specifications for subproblems in such a way that solutions for the subproblems can be assembled (via the scheme) into a solution for Π . A design strategy then is a way of generating an instance of a scheme which satisfies a given specification. Any program scheme admits a number of design strategies. Dershowitz and Manna [4] have presented some strategies for designing program sequences, if-then-else statements, and loops.

We have found three design strategies for divide and conquer algorithms. Each attempts to derive specifications for subalgorithms which satisfy the conditions of Theorem 1. If successful then any operators which satisfy these derived specifications can be assembled into a divide and conquer algorithm satisfying the given specification. The key difficulty is to ensure that the derived specifications satisfy the separability condition, so each design strategy concentrates on this goal.

The first design strategy, called DS1, can be described as follows.

DS1) First choose a simple decomposition algebra as E and choose simple known functions for the auxiliary functions, then use the separability condition to reason backwards towards output conditions and to reason forwards towards input conditions for the operators in T .

To see how we reason towards specifications for the operators in T , suppose that we have selected a Σ^{-1} -algebra E and chosen simple known functions f_s for $s \in S - \hat{S}$ and let the given problem be $\Pi = \langle D, R, I, O \rangle$. We show how to derive output conditions for σ_{i_T} for some $i \in \underline{r}$. First use

$$\sigma_{i_E}: x_0 = \langle x_1, \dots, x_{n_i} \rangle \text{ as } O_{i_E}: \langle z_0, z_1, \dots, z_{n_i} \rangle,$$

$$f_{w_{ij}}: x_j = z_j \text{ as } P_{w_{ij}}: \langle x_j, z_j \rangle \text{ for } 1 \leq j \leq n_i, w_{ij} \neq \hat{S}, \text{ and}$$

$$O: \langle x, z \rangle \text{ as } P_{\hat{S}}: \langle x, z \rangle,$$

and create the following formula

$$\begin{aligned} & \forall \langle x_0, x_1, \dots, x_n \rangle \in E^{\hat{S}w_i} \quad \forall \langle z_0, z_1, \dots, z_n \rangle \in T^{\hat{S}w_i} \\ & [O_{i_E}: \langle x_0, x_1, \dots, x_{n_i} \rangle \bigwedge_{j \in \underline{r}} P_{w_{ij}}: \langle x_j = z_j \rangle \Rightarrow P_{\hat{S}}: \langle x_0, z_0 \rangle]. \end{aligned} \quad (4.1)$$

This formula differs from the separability condition only in that the hypothesis $O_{i_T}: \langle z_0, z_1, \dots, z_n \rangle$ is missing. We desire to establish the separability condition so that we can apply Theorem 1 to show that the program we construct satisfies its specification. We know that O_{i_T} it is a relation on the variables z_0, z_1, \dots, z_{n_i} . Our technique is to reason backwards from the consequent always trying to reduce it to relations expressed in terms of the variables z_0, z_1, \dots, z_{n_i} . If we can show that the assumption of an additional hypothesis of the form

$$Q: \langle z_0, z_1, \dots, z_{n_i} \rangle$$

allows us to prove (4.1), i.e., if we can show that

$$\forall \langle x_0, x_1, \dots, x_n \rangle \in E^{\text{Swi}} \quad \forall \langle z_0, z_1, \dots, z_n \rangle \in T^{\text{Swi}} \\ [O_{i_E} : \langle x_0, x_1, \dots, x_{n_1} \rangle \wedge \bigwedge_{j \in I} P_{wi_j} : \langle x_j = z_j \rangle \wedge Q : \langle z_0, z_1, \dots, z_{n_1} \rangle \Rightarrow P_s : \langle x_0, z_0 \rangle]$$

then we take Q as the output condition O_{i_T} since the separability condition is satisfied by this choice of O_{i_T} . Formal systems for performing this kind of deduction are presented in [12,13]. We shall proceed a little less formally here, making use of our intuition for guidance.

We can also use (4.1) to obtain input conditions for our composition operators. The input condition for σ_{i_T} is some relation on z_1, \dots, z_{n_1} which can be expected to hold when σ_{i_T} is invoked. Suppose that by reasoning forwards from the relations established by the decomposition operator and the component functions we infer a relation $Q' : \langle z_1, \dots, z_{n_1} \rangle$, i.e., that

$$\forall \langle x_0, x_1, \dots, x_n \rangle \in E^{\text{Swi}} \quad \forall \langle z_0, z_1, \dots, z_n \rangle \in T^{\text{Swi}} \\ [O_{i_E} : \langle x_0, x_1, \dots, x_{n_1} \rangle \wedge \bigwedge_{j \in I} P_{wi_j} : \langle x_j, z_j \rangle \Rightarrow Q' : \langle z_1, \dots, z_{n_1} \rangle].$$

Then we take Q' as an input condition to σ_{i_T} .

The other two design strategies are variations on DS1 and use the separability condition in an analogous manner.

DS2) First choose a simple composition algebra as T , second, choose simple known functions for the auxiliary functions, then use the separability condition to solve for the input and output conditions for the operators in E . An input condition for the decomposition operator is found by determining conditions under which a feasible output exists.

DS3) First choose a simple decomposition Σ^{-1} -algebra as E and choose a simple composition Σ -algebra as T , then use the separability condition to reason backwards towards output conditions and to reason forwards towards input conditions for the auxiliary functions.

In each of these design strategies we must find a suitable well-founded ordering on the input domain in order to ensure program termination. Also, the guards are chosen to reflect the domain of definition of the decomposition operators.

4.2 Design of a Selection Sort Algorithm

Suppose we are given the following specification for sorting a list of natural numbers

$$\text{SORT: } x = z \text{ such that } \text{Bag: } x = \text{Bag: } z \wedge \text{Ordered: } z \\ \text{where } \text{Sort: } \text{LIST}(\text{IN}) \rightarrow \text{LIST}(\text{IN}).$$

Here "Bag: $x = \text{Bag: } z$ " asserts that the multiset (bag) of elements in the list z is the same as the multiset of elements in x . Ordered is a predicate which holds when applied to a list whose elements are in nondecreasing order.

The selection sort algorithm presented in Figure 4 will be derived using design strategy DS2. Note that Ssort makes use of the composition algebra $A = \langle \{\text{LIST}(\text{IN}), \text{IN}\}, \{\text{Nil}, \text{Cons}\} \rangle$ discussed in Section 2.2. In choosing A as the composition algebra it is not obvious ahead of time that a decomposition algebra can be found which works with A to solve the SORT problem. This choice of algebra should be regarded as a tentative hypothesis about how sorted lists can be composed. The sort set of A is $S = \{c, s\}$ where $A_s = \text{LIST}(\text{IN})$ and $A_c = \text{IN}$. The operator Nil has type $\langle \lambda, s \rangle$ and operator Cons has type $\langle c s, s \rangle$, $\text{Nil: } A^\lambda \rightarrow A_s$, and $\text{Cons: } A^{cs} \rightarrow A_s$.

Naming our desired program Ssort we have at this point,

$$E_s = \text{LIST}(\text{IN}), T_s = \text{LIST}(\text{IN}), T_c = \text{IN}$$

$$J_s \leftrightarrow \text{TRUE},$$

$$P_s: \langle x, z \rangle \leftrightarrow \text{Bag: } x = \text{Bag: } z \wedge \text{Ordered: } z,$$

$$O1_T: \langle \langle \rangle, z \rangle \leftrightarrow z = \text{nil},$$

$$O2_T: \langle z_0, b, z_1 \rangle \leftrightarrow \text{Cons: } \langle b, z_1 \rangle = z_0,$$

$$f_s \text{ is Ssort.}$$

It remains to determine input and output conditions J_c and P_c for the auxiliary function f_c , the domain E_c , and the output conditions $O1_E$ and $O2_E$ for the decomposition operators.

Our first step towards determining $O2_E$ is to instantiate the separability condition as far as possible thus obtaining

$$\forall \langle x_0, \langle a, x_1 \rangle \rangle \in \text{LIST}(\text{IN}) \times (E_c \times \text{LIST}(\text{IN})) \quad \forall \langle z_0, \langle b, z_1 \rangle \rangle \in \text{LIST}(\text{IN}) \times (\text{IN} \times \text{LIST}(\text{IN}))$$

```

Ssort:x = if
    x=nil → Nil·Id◇·liN:x []
    x≠nil → Cons·(IdX Ssort)·Select:x
fi

Select:x = if
    Rest:x=nil → Compose1·Id·snoC:x []
    Rest:x≠nil → Compose2·(IdX Select)·SnoC:x
fi

Compose1:v = <v,nil>

Compose2:<v1,<v2,z>> = if
    v1 ≤ v2 → <v1,Cons:<v2,z>> []
    v1 ≥ v2 → <v2,Cons:<v1,z>>
fi

```

Figure 4: A Selection Sort Program

$$\begin{aligned}
 & [O2_E: \langle x_0, \langle a, x_1 \rangle \rangle \wedge P_C: \langle a, b \rangle \wedge \text{Bag}: x_1 = \text{Bag}: z_1 \wedge \text{Ordered}: z_1 \wedge \text{Cons}: \langle b, z_1 \rangle = z_0 \\
 & \Rightarrow \text{Bag}: x_0 = \text{Bag}: z_0 \wedge \text{Ordered}: z_0] \quad (4.2)
 \end{aligned}$$

To construct this formula we have made the following substitutions into the separability condition of Theorem 1:

1. replace w2 by c_s
2. replace E_s and T_s by LIST(IN)
3. replace E^{c_s} by E_CXLIST(IN) and T^{c_s} by INXLIST(IN)
4. replace P_s:<x,z> by Bag:x=Bag:z ∧ Ordered:z
5. replace σ_T:<b,z₁> by Cons:<b,z₁>

Since we desire to have the separability condition hold in order to apply Theorem 1 we evidently must try to find values for E_C, P_C, and O2_E which allow us to prove (4.2).

In order to determine $O2_E$ we attempt to reduce (4.2) to a formula dependent on the variables x_0 , a , and x_1 only. The consequent is the conjunction of two atomic formulas so we can tackle them separately. Consider first

$$\text{Bag}:x_0 = \text{Bag}:z_0. \quad (4.3)$$

This is equivalent to

$$\text{Bag}:x_0 = \text{Bag}:\text{Cons}:\langle b, z_1 \rangle$$

since $\text{Cons}:\langle b, z_1 \rangle = z_0$ is a hypothesis. The fact

$$\text{Bag} \cdot \text{Cons}:\langle u, y \rangle = \text{Add}:\langle b, \text{Bag}:y \rangle$$

allows us to reduce the goal to

$$\text{Bag}:x_0 = \text{Add}:\langle b, \text{Bag}:z_1 \rangle.$$

Then since

$$\text{Bag}:x_1 = \text{Bag}:z_1$$

is a hypothesis we further reduce to

$$\text{Bag}:x_0 = \text{Add}:\langle b, \text{Bag}:x_1 \rangle.$$

This last relation is almost expressed in terms of variables required by $O2_E$. Let us assume $a = b$ and thus let $E_c = \text{IN}$, $J_c:x \Leftrightarrow \text{TRUE}$, $P_c:\langle a, b \rangle \Leftrightarrow a = b$, and let f_c be Id. This finally reduces (4.3) to

$$\text{Bag}:x_0 = \text{Add}:\langle a, \text{Bag}:x_1 \rangle. \quad (4.4)$$

In other words, if we had (4.4) and $a = b$ as additional hypotheses then we could establish our original goal (4.3). We will use (4.4) in the output condition $O2_E$.

Consider now the second goal

$$\text{Ordered}:z_0 \quad (4.5)$$

which via the hypotheses $\text{Cons}:\langle b, z_1 \rangle = z_0$ and $a = b$ reduces to

$$\text{Ordered} \cdot \text{Cons}:\langle a, z_1 \rangle.$$

The fact

$$u \leq \text{Bag}:y \wedge \text{Ordered}:y \Leftrightarrow \text{Ordered} \cdot \text{Cons}:\langle u, y \rangle$$

can be used to produce the equivalent goal

$$a \leq \text{Bag}:z_1 \wedge \text{Ordered}:z_1.$$

Now $\text{Ordered}:z_1$ is a hypothesis and thus is assumed to hold. The remaining subgoal can be transformed via the hypothesis $\text{Bag}:x_1 = \text{Bag}:z_1$ to

$$a \leq \text{Bag}:x_1.$$

We have reduced (4.5) to a subgoal which is expressed in terms of the variables

required by $O2_E$. By reasoning backwards we have shown above that if

$$a \leq \text{Bag}:x_1 \wedge \text{Bag}:x_0 = \text{Add}:<a, \text{Bag}:x_1> \quad (4.6)$$

holds then we can establish (4.2). We take (4.6) as $O2_E$.

Before constructing the specification for $\sigma 2_E$ we construct a well-founded ordering on $E_s = \text{LIST}(\text{IN})$. By Proposition 1 we can construct one based on a mapping from $\text{LIST}(\text{IN})$ to IN . The known function Length maps $\text{LIST}(\text{IN})$ to IN so define

$$x_0 \succ x_1 \text{ iff } \text{Length}:x_0 > \text{Length}:x_1.$$

By Proposition 1 $\langle E_s, \succ \rangle$ is a well-founded set.

Using (4.6) as $O2_E$ and this well-founded ordering on $\text{LIST}(\text{IN})$ we create the following specification for $\sigma 2_E$ in accord with condition (1) of Theorem 1.

$$\begin{aligned} \sigma 2_E:x_0 = & \langle a, x_1 \rangle \text{ such that } a \leq \text{Bag}:x_1 \wedge \text{Bag}:x_0 = \text{Add}:<a, \text{Bag}:x_1> \wedge \\ & \text{Length}:x_0 > \text{Length}:x_1 \\ & \text{where } \sigma_E:\text{LIST}(\text{IN}) \rightarrow \text{IN} \times \text{LIST}(\text{IN}) \end{aligned}$$

By inspection we see that there is no feasible output when the input is nil so we add the input condition " $x \neq \text{nil}$ " obtaining

$$\begin{aligned} \sigma 2_E:x_0 = & \langle a, x_1 \rangle \text{ such that } x_0 \neq \text{nil} \Rightarrow \text{Bag}:x_0 = \text{Add}:<a, \text{Bag}:x_1> \wedge \\ & a \leq \text{Bag}:x_1 \wedge \text{Length}:x_0 > \text{Length}:x_1 \\ & \text{where } \sigma_E:\text{LIST}(\text{IN}) \rightarrow \text{IN} \times \text{LIST}(\text{IN}). \end{aligned}$$

In [13] we show how to derive the input condition for decomposition operators by formal means. In the next section we derive a divide and conquer algorithm, called *Select*, for this problem.

From the input condition of *Select* we obtain the guard $x \neq \text{nil}$. The intended algorithm at this point has the form:

```
Ssort:x = if
    q1:x → Nil.fλ.σ1E:x []
    x ≠ nil → Cons.(Id X Ssort).Select:x
fi.
```

The construction of a specification for $\sigma 1_E$ is similar. First, we instantiate the separability condition obtaining

$$\forall x_0 \in \text{LIST}(\text{IN}) \quad \forall z_0 \in \text{LIST}(\text{IN})$$

$$[Ol_E:x_0 \wedge Nil:\diamond = z_0 \Rightarrow Bag:x_0 = Bag:z_0 \wedge Ordered:z_0] \quad (4.7)$$

In creating this formula we have replaced

w_1 by λ

E_s and T_s by $LIST(IN)$

P_s by $Bag:x_0 = Bag:z_0 \wedge Ordered:z_0]$

σ_{l_T} by Nil

and performed some simplifications.

Again we treat the two conjuncts of the goal separately. Since z_0 is nil then the goal $Ordered:z_0$ holds. The other goal

$$Bag:z_0 = Bag:x_0$$

is equivalent to

$$x_0 = nil$$

since $z_0 = nil$. We use " $x_0 = nil$ " as the output condition of Ol_E and create the specification

$$\begin{aligned} \sigma_{l_E}:x_0 &= z \text{ such that } x_0 = nil \\ \text{where } \sigma_{l_E}:LIST(IN) &\rightarrow \{\langle \rangle\}. \end{aligned}$$

The function lin satisfies this specification.

Putting together all of the operators derived above, we obtain the following selection sort program:

```
Ssort:x ≡ if
    x=nil → Nil·Id◇·lin:x []
    x≠nil → Cons·(IdX Ssort)·Select:x
fi
```

which can be simplified to

```
Ssort:x ≡ if
    x=nil → x []
    x≠nil → Cons·(IdX Ssort)·Select:x
fi
```

4.3 Synthesis of Select

In the previous section we derived the specification

$\text{Select}:x_0 = \langle a, x_1 \rangle \text{ such that } x_0 \neq \text{nil} \Rightarrow \text{Bag}:x_0 = \text{Add}: \langle a, \text{Bag}:x_1 \rangle \wedge$
 $a \leq \text{Bag}:x_1 \wedge \text{Length}:x_0 > \text{Length}:x_1.$
 where $\text{Select}:\text{LIST}(\text{IN}) \rightarrow \text{IN} \times \text{LIST}(\text{IN})$

The synthesis of Select proceeds according to the design strategy DS2. First, we choose a simple decomposition algebra for the input domain - the set of non-nil lists of natural numbers. The algebra $A = \langle \{\text{IN}, \text{LIST}(\text{IN})\}, \{\text{tsiL}, \text{snoC}\} \rangle$ is satisfactory since all non-nil lists can be decomposed into non-nil lists and natural numbers by tsiL and snoC. The sort set is $S = \{c, \hat{s}\}$, tsiL has type $\langle \hat{s}, c \rangle$, and snoC has type $\langle \hat{s}, c\hat{s} \rangle$. We have

$E_c = \text{N},$
 $E_{\hat{s}} = \text{LIST}(\text{IN}), T_{\hat{s}} = \text{IN} \times \text{LIST}(\text{IN}),$
 $J_{\hat{s}}:x_0 \Leftrightarrow x_0 \neq \text{nil},$
 $P_{\hat{s}}:\langle x_0, \langle a, x_1 \rangle \rangle \Leftrightarrow \text{Bag}:x_0 = \text{Add}: \langle a, \text{Bag}:x_1 \rangle \wedge a \leq \text{Bag}:x_1 \wedge \text{Length}:x_0 > \text{Length}:x_1$

$\sigma 1_E$ is tsiL, and $\sigma 2_E$ is snoC.

tsiL is defined when $\text{Rest}:x = \text{nil}$ so this condition is used as q_1 . snoC will decompose a non-nil list x into a number and a non-nil list when $\text{Rest}:x \neq \text{nil}$, so we take this condition as q_2 . Our intended algorithm now has the form

$\text{Select}:x_0 = \text{if}$
 $\quad \text{Rest}:x_0 = \text{nil} \rightarrow \sigma 1_T \cdot f_c \cdot \text{tsiL}:x_0 \quad []$
 $\quad \text{Rest}:x_0 \neq \text{nil} \rightarrow \sigma 2_T \cdot (f_c \times \text{Select}) \cdot \text{snoC}:x_0$
 fi

It remains to determine the output domain T_c , the input and output conditions J_c and P_c for the auxiliary function f_c , and the composition operators $\sigma 1_T$ and $\sigma 2_T$.

$E_{\hat{s}} = \text{LIST}(\text{IN})$ is made a well-founded set exactly as in the previous example by defining $x_0 \succ x_1$ iff $\text{Length}:x_0 > \text{Length}:x_1$. snoC and tsiL clearly preserve this ordering.

In pursuit of an output condition for $\sigma 2_T$ (a relation dependent on the variables a_0, z_0, v, a_1 , and z_1), we first instantiate the separability condition with the result

$\forall \langle \langle a_0, z_0 \rangle, \langle v, \langle a_1, z_1 \rangle \rangle \rangle \in (\text{IN} \times \text{LIST}(\text{IN})) \times (T_c \times (\text{IN} \times \text{LIST}(\text{IN})))$
 $\forall \langle x_0, \langle u, x_1 \rangle \rangle \in \text{LIST}(\text{IN}) \times (\text{IN} \times \text{LIST}(\text{IN}))$
 $[\text{snoC}:x_0 = \langle u, x_1 \rangle \wedge \text{Bag}:x_1 = \text{Add}: \langle a_1, \text{Bag}:z_1 \rangle \wedge a_1 \leq \text{Bag}:z_1 \wedge$
 $\text{Length}:x_1 > \text{Length}:z_1 \wedge P_c:\langle u, v \rangle \wedge \sigma 2_T:\langle \langle a_0, z_0 \rangle, \langle v, \langle a_1, z_1 \rangle \rangle \rangle]$

$$\Rightarrow \text{Bag}:x_0 = \text{Add}:\langle a_0, z_0 \rangle \wedge a_0 \leq \text{Bag}:z_0 \wedge \text{Length}:x_0 > \text{Length}:z_0]. \quad (4.8)$$

To create this formula the following substitutions were made

c_s replaces w_2

$\text{LIST}(\text{IN})$ replaces E_s and $\text{IN} \setminus \text{LIST}(\text{IN})$ replaces T_s

IN replaces E_c

$\text{snoc}:x_0 = \langle u, x_1 \rangle$ replaces $\sigma_{2E}:\langle x_0, x_1, x_2 \rangle$

$\text{Bag}:x_1 = \text{Add}:\langle a_1, \text{Bag}:z_1 \rangle \wedge a_1 \leq \text{Bag}:z_1 \wedge \text{Length}:x_1 > \text{Length}:z_1$
replaces $P_s:\langle x_1, \langle a_1, z_1 \rangle \rangle$

Again we consider the goals in (4.8) one at a time. The goal

$$a_0 \leq \text{Bag}:z_0$$

is already expressed in the form we desire, so we can use it in σ_{2T} . Consider the goal

$$\text{Bag}:x_0 = \text{Add}:\langle a_0, z_0 \rangle.$$

We have

$$\begin{aligned} \text{Bag}:x_0 &= \text{Bag} \cdot \text{Cons}:\langle u, x_1 \rangle \quad (\text{by hypothesis}) \\ &= \text{Add}:\langle u, \text{Bag}:x_1 \rangle \\ &= \text{Add}:\langle u, \text{Add}:\langle a_1, z_1 \rangle \rangle \quad (\text{by hypothesis}) \end{aligned}$$

Suppose that we let $u=v$ and thus let $T_c = \text{IN}$, $P_c:\langle u, v \rangle \Leftrightarrow u=v$, and f_c be Id. We have

$$\text{Add}:\langle v, \text{Add}:\langle a_1, z_1 \rangle \rangle = \text{Add}:\langle a_0, z_0 \rangle.$$

This condition is expressed in the desired variables so we use it in σ_{2T} . Finally, consider the goal

$$\text{Length}:x_0 > \text{Length}:z_0. \quad (4.9)$$

In the following derivation we use $\text{Card}:x$ to denote the cardinality of the bag x . We then have

$$\begin{aligned} \text{Length}:x_0 &= \text{Length} \cdot \text{Cons}::\langle u, x_1 \rangle \\ &= 1 + \text{Length}:x_1 \\ &= 1 + \text{Card} \cdot \text{Add}:\langle a_1, \text{Bag}:z_1 \rangle && (\text{using hypothesis} \\ &&& \text{Bag}:x_1 = \text{Add}:\langle a_1, \text{Bag}:z_1 \rangle) \\ &= 2 + \text{Card} \cdot \text{Bag}:z_1 \\ &= 2 + \text{Length}:z_1. \end{aligned}$$

Thus we have reduced (4.9) to

$$2 + \text{Length}:z_1 > \text{Length}:z_0.$$

Putting all these conditions together we obtain

$$\begin{aligned} \text{Add}:<v, \text{Add}:<a_1, \text{Bag}:z_1>> &= \text{Add}:<a_0, \text{Bag}:z_0> \wedge \\ a_0 \leq \text{Bag}:z_0 &\wedge 2 + \text{Length}:z_1 > \text{Length}:z_0 \end{aligned}$$

and use it as $O2_T$. We derive an input condition by reasoning forwards from

$$\text{snoc}:x_0 = <u, x_1> \wedge \text{Bag}:x_1 = \text{Add}:<a_1, \text{Bag}:z_1> \wedge a_1 \leq \text{Bag}:z_1 \wedge \text{Length}:x_1 > \text{Length}:z_1 \wedge u = v$$

towards a relation expressed in terms of the variables v , a_1 , and z_1 . The only useful inference seems to be

$$a_1 \leq \text{Bag}:z_1$$

so we take this as the input condition and form the specification

$$\begin{aligned} \sigma 2_T: <v, <a_1, z_1>> &= <a_0, z_0> \text{ such that } a_1 \leq \text{Bag}:z_1 \Rightarrow a_0 \leq \text{Bag}:z_0 \wedge \\ \text{Add}:<v, \text{Add}:<a_1, \text{Bag}:z_1>> &= \text{Add}:<a_0, \text{Bag}:z_0> \wedge 2 + \text{Length}:z_1 > \text{Length}:z_0 \\ \text{where } \sigma 2_T: \text{IN} \times (\text{IN} \times \text{LIST}(\text{IN})) &\rightarrow \text{IN} \times \text{LIST}(\text{IN}) \end{aligned}$$

A conditional program, call it Compose2 , can be constructed satisfying this specification.

$$\begin{aligned} \text{Compose2}:<v, <a_1, z_1>> &= \text{if} \\ v \leq a_1 &\rightarrow <v, \text{Cons}:<a_1, z_1>>[] \\ v \geq a_1 &\rightarrow <a_1, \text{Cons}:<v, z_1>> \\ \text{fi} \end{aligned}$$

We construct $O1_T$ in a similar manner. The separability condition is partially instantiated yielding

$$\begin{aligned} \forall <<a_0, z_0>, v> \in \text{IN} \times \text{LIST}(\text{IN}) \times \text{IN} \quad \forall <x_0, u> \in \text{LIST}(\text{IN}) \times \text{IN} \\ [\text{tsiL}:x_0 = u \wedge u = v \\ \Rightarrow \text{Bag}:x_0 = \text{Add}:<a_0, \text{Bag}:z_0> \wedge a_0 \leq \text{Bag}:z_0 \wedge \text{Length}:x_0 > \text{Length}:z_0]. \end{aligned} \quad (4.9)$$

Dealing first with the goal

$$\text{Bag}:x_0 = \text{Add}:<a_0, \text{Bag}:z_0>$$

we have

$$\text{Bag}:x_0 = \{u\} = \{v\}$$

thus

$$\{v\} = \text{Add}: \langle a_0, \text{Bag}: z_0 \rangle$$

or equivalently

$$a_0 = v \wedge z_0 = \text{nil}.$$

Again the second goal $a_0 \leq \text{Bag}: z_0$ is already reduced to the desired form. Consider now the final goal

$$\text{Length}: x_0 > \text{Length}: z_0.$$

We have $\text{Length}: x_0 = 1$ thus the goal must reduce to

$$\text{Length}: z_0 = 0$$

or equivalently, $z_0 = \text{nil}$.

Putting together all these conditions we obtain

$$\text{Ol}_T: \langle z_0, v \rangle \iff z_0 = \text{nil} \wedge a_0 = v$$

and create the specification

$$\begin{aligned} \sigma_{1_T}: v = \langle a, z \rangle \text{ such that } z = \text{nil} \wedge a = v. \\ \text{where } \sigma_{1_T}: \text{LIST}(\text{IN}) \rightarrow \text{IN} \times \text{LIST}(\text{IN}). \end{aligned}$$

The function Compose1 is easily shown to satisfy this specification:

$$\text{Compose1}: v = \langle v, \text{nil} \rangle.$$

The functions derived above are assembled into the following program:

```
Select: x0 = if
    Rest: x0 = nil → Compose1 · Id◇ · tsil: x0 []
    Rest: x0 ≠ nil → Compose2 · (Id × Select) · snoc: x0
fi
```

The complete selection sort program derived in this section is listed in Figure 4. It can be transformed into the simpler program listed in Figure 1.

5. More Examples

5.1. Cartesian Product of Two Sets

In this section we illustrate the design of a divide and conquer algorithm using design strategy DS3. The problem of forming the cartesian product of two sets can be specified by

$CART_PROD: \langle x, x' \rangle = z$ such that $z = \{ \langle a, b \rangle \mid a \in x \text{ and } b \in x' \}$
 where $CART_PROD: SET(IN) \times SET(IN) \rightarrow SET(IN \times IN)$.

Here $SET(R)$ denotes the data type of finite sets whose elements belong to the data type R .

First, we choose a decomposition algebra on $SET(IN) \times SET(IN)$ and then a composition algebra on $SET(IN \times IN)$. A simple decomposition algebra on sets is easily found:

$$A_1 = \langle \{SET(IN), IN\}, \{Split, ihP\} \rangle$$

where

$$A_{1_s} = SET(IN)$$

$$A_{1_c} = IN$$

$$\sigma_{1_{A_1}} = ihP: SET(R) \rightarrow \{ \langle \rangle \} \quad (\text{type } \langle \lambda, \hat{s} \rangle)$$

$$\sigma_{2_{A_1}} = Split: SET(R) \rightarrow R \times SET(R) \quad (\text{type } \langle c\hat{s}, \hat{s} \rangle).$$

ihP decomposes the empty set into the 0-tuple $\langle \rangle$ and $Split$ decomposes a nonempty set into an element and the remainder of the set. ihP is defined only on the empty set and $Split$ is defined only on nonempty sets so together these operators decompose every finite set.

However, our input domain is 2-tuples of sets. We shall apply the above decomposition operators to the first component of the tuple and leave the second unchanged. The result is the Σ^{-1} -decomposition algebra

$$A_2 = \langle \{IN \times SET(IN), SET(IN) \times SET(IN)\}, \{ihP \cdot \underline{1}, Trans \cdot (Split \times Id_2)\} \rangle.$$

where

$$A_{2_s} = SET(IN) \times SET(IN),$$

$$A_{2_c} = IN \times SET(IN),$$

$$\sigma_{1_{A_2}} = ihP \cdot \underline{1}: SET(IN) \times SET(IN) \rightarrow \{ \langle \rangle \} \quad (\text{type } \langle \lambda, \hat{s} \rangle),$$

$$\sigma_{2_{A_2}} = Trans \cdot (Split \times Id_2): SET(IN) \times SET(IN) \rightarrow (IN \times SET(IN)) \times (SET(IN) \times SET(IN)) \\ (\text{type } \langle c\hat{s}, \hat{s} \rangle).$$

$\sigma_{2_{A_2}}$ makes use of two new functions. The function Id_2 returns a 2-tuple containing copies of its input, i.e., $Id_2: x = \langle x, x \rangle$. The function $Trans$ transposes a tuple of tuples as follows

Trans: $\langle x_1, \dots, x_n \rangle = \langle y_1, \dots, y_m \rangle$
 where $x_i = \langle x_{i1}, \dots, x_{im} \rangle$ and $y_j = \langle x_{1j}, \dots, x_{nj} \rangle$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. For example,

$$\text{Trans}: \langle \langle 1, 2, 3 \rangle, \langle 4, 5, 6 \rangle \rangle = \langle \langle 1, 4 \rangle, \langle 2, 5 \rangle, \langle 3, 6 \rangle \rangle.$$

σ_{2A2} behaves as follows on input $\langle \{1, 2, 3\}, \{4, 5\} \rangle$:

$$\begin{aligned} \text{Trans} \cdot (\text{Split} \times \text{Id}_2) : \langle \{1, 2, 3\}, \{4, 5\} \rangle &= \text{Trans} : \langle \langle 1, \{2, 3\} \rangle, \langle \{4, 5\}, \{4, 5\} \rangle \rangle \\ &= \langle \langle 1, \{4, 5\} \rangle, \langle \{2, 3\}, \{4, 5\} \rangle \rangle. \end{aligned}$$

Before choosing a composition algebra for T we must decide what can the auxiliary output type T_c be given that E_c is $\text{IN} \times \text{SET}(\text{IN})$. Since E_c appears to be a slightly modified form of E_s ($= \text{SET}(\text{IN}) \times \text{SET}(\text{IN})$) we might conjecture that the auxiliary function f_c is similar to the principal function f_s and thus use $\text{SET}(\text{IN} \times \text{IN})$ as T_c . The composition operator σ_{2T} then is some mapping from $\text{SET}(\text{IN} \times \text{IN}) \times \text{SET}(\text{IN} \times \text{IN})$ to $\text{SET}(\text{IN} \times \text{IN})$ - we can use the set-union operator Union. σ_{1T} is some mapping from $\{\langle \rangle\}$ to $\text{SET}(\text{IN} \times \text{IN})$ - we can use the function Phi, which maps the 0-tuple into the empty set.

So far we have developed the program structure

```
CP:  $\langle x, x' \rangle \Rightarrow$  if
     $x = \{\} \rightarrow \text{Phi} \cdot \text{Id}_\square \cdot \text{ihP} \cdot \text{I} : \langle x, x' \rangle \square$ 
     $x \neq \{\} \rightarrow \text{Union} \cdot (f_c \times \text{CP}) \cdot \text{Trans} \cdot (\text{Split} \times \text{Id}_2) : \langle x, x' \rangle \square$ 
fi.
```

In order to determine a specification for f_c we create the following instance of the separability condition

$$\begin{aligned} \forall \langle \langle x_0, x'_0 \rangle, \langle a, x'_1 \rangle, \langle x_2, x'_2 \rangle \rangle \in (\text{SET}(\text{IN}) \times \text{SET}(\text{IN})) \times (\text{IN} \times \text{SET}(\text{IN})) \times (\text{SET}(\text{IN}) \times \text{SET}(\text{IN})) \\ \forall \langle z_0, z_1, z_2 \rangle \in \text{SET}(\text{IN} \times \text{IN}) \times \text{SET}(\text{IN} \times \text{IN}) \times \text{SET}(\text{IN} \times \text{IN}) \\ [\text{Split} : x_0 = \langle a, x_2 \rangle \wedge x'_1 = x'_0 \wedge x'_2 = x'_0 \wedge P_c : \langle \langle a, x'_1 \rangle, z_1 \rangle \wedge \\ z_2 = \{ \langle u, v \rangle \mid u \in x_2 \text{ and } v \in x'_2 \} \wedge \\ z_0 = \text{Union} : \langle z_1, z_2 \rangle \Rightarrow z_0 = \{ \langle u, v \rangle \mid u \in x_0 \text{ and } v \in x'_0 \}]. \end{aligned} \quad (5.1)$$

Since we are trying to reason backwards to an expression for $P_c : \langle \langle a, x'_1 \rangle, z_1 \rangle$ we seek to reduce the goal to a relation over the variables a , x'_1 , and z_1 . Consider the goal

$$z_0 = \{ \langle u, v \rangle \mid u \in x_0 \text{ and } v \in x'_0 \}. \quad (5.2)$$

The set expression on the right hand side can be transformed as follows.

$$\begin{aligned}
\{ \langle u, v \rangle \mid u \in x_0 \text{ and } v \in x'_0 \} &= \{ \langle u, v \rangle \mid u \in \text{Add} : \langle a, x_2 \rangle \text{ and } v \in x'_0 \} \\
&\quad (\text{since Split} : x = \langle a, y \rangle) \\
&= \{ \langle u, v \rangle \mid (u = a \text{ or } u \in x_2) \text{ and } v \in x'_0 \} \\
&= \text{Union} : \langle \{ \langle u, v \rangle \mid u = a \text{ and } v \in x'_0 \}, \{ \langle u, v \rangle \mid u \in x_2 \text{ and } v \in x'_0 \} \rangle \\
&= \text{Union} : \langle \{ \langle u, v \rangle \mid u = a \text{ and } v \in x'_1 \}, \{ \langle u, v \rangle \mid u \in x_2 \text{ and } v \in x'_2 \} \rangle \\
&\quad (\text{since } x'_1 = x'_0 \text{ and } x'_2 = x'_0) \\
&= \text{Union} : \langle \{ \langle u, v \rangle \mid u = a \text{ and } v \in x'_1 \}, z_2 \rangle. \\
&\quad (\text{since } z_0 = \{ \langle u, v \rangle \mid u \in x_0 \text{ and } v \in x'_0 \}).
\end{aligned}$$

Using the hypothesis $z_0 = \text{Union} : \langle z_1, z_2 \rangle$ we reduce (5.2) to

$$\text{Union} : \langle z_1, z_2 \rangle = \text{Union} : \langle \{ \langle u, v \rangle \mid u = a \text{ and } v \in x'_1 \}, z_2 \rangle$$

which holds if

$$z_1 = \{ \langle u, v \rangle \mid u = a \text{ and } v \in x'_0 \} \quad (5.3)$$

holds. So if we take (5.3) as an additional hypothesis then (5.1) holds. We take (5.3) as our output condition for f_c and create the specification

$$\begin{aligned}
\text{CP_aux} : \langle a, x \rangle = z \text{ such that } z &= \{ \langle u, v \rangle \mid u = a \text{ and } v \in x \} \\
\text{CP_aux} : \text{IN} \times \text{SET}(\text{IN}) &\rightarrow \text{SET}(\text{IN}) \times \text{SET}(\text{IN}).
\end{aligned}$$

A divide and conquer algorithm for this problem can easily be constructed using design strategy DSI (along the same lines as Ssort). The complete algorithm for producing the cartesian product of two sets is listed in Figure 5. The reader can easily find several ways to simplify CP and CP_aux without affecting their correctness.

5.2 Evaluating a Proposition

In this section we present a divide and conquer algorithm for evaluating a proposition. It provides an example of a more complex signature and illustrates a programming style suggested by our treatment of divide and conquer algorithms. Given a well-formed proposition F and an interpretation I the problem is to compute the truth value of F under I . Relevant portions of the abstract data types for propositions, interpretations, and truth values are presented below.

A data type PROP representing well-formed propositions can be described abstractly as follows. Let LETTERS be a set of symbols called letters. PROP is generated from LETTERS using the constructors

```

CP:<x,x'> = if
    x = {} → Phi·Id◇·ihP·1:<x,x'> []
fi.
    x ≠ {} → Union·(CP_aux X CP)·Trans·(Split X Id2):<x,x'> []

CP_aux:<a,x> = if
    x = {} → Phi·Id◇·ihP·2:<a,x> []
fi.
    x ≠ {} → Add·(Id X CP_aux)·Trans·(Id2 X Split):<a,x> []

```

Figure 5. Forming the Cartesian Product of Two Sets.

Compose_atom:LETTER → PROP, which converts a letter into an atomic proposition,
 Compose_neg:PROP → PROP, which forms the negation of a proposition,
 Compose_conj:PROP X PROP → PROP, which forms the conjunction of two propositions,
 Compose_disj:PROP X PROP → PROP, which forms the disjunction of two propositions.

In other words we have

<{PROP,LETTERS}, {Compose_atom, Compose_neg, Compose_conj, Compose_disj}>

as a composition algebra for PROP. Each of these constructors are uniquely invertible and we have the corresponding decomposition algebra

<{PROP,LETTERS}, {Decompose_atom, Decompose_neg, Decompose_conj, Decompose_disj}>

where

Decompose_atom:PROP → LETTER, which decomposes an atomic proposition into its constituent letter,

Decompose_neg:PROP → PROP, which decomposes a negation into its constituent proposition,

Decompose_conj:PROP → PROP X PROP, which decomposes a conjunction into its constituent propositions, and

Decompose_disj:PROP → PROP X PROP, which decomposes a disjunction into its constituent propositions.

These decomposition operators are defined when the predicates Atom, Neg, Conj, Disj are true respectively. For example, Atom:F holds exactly when Decompose_atom:F = oc for some $oc \in \text{LETTER}$. We also have $F = \text{Compose_atom:oc}$. Similarly, Conj:F holds iff Decompose_conj:F = $\langle G, H \rangle$ for some $G, H \in \text{PROP}$ and thus $F = \text{Compose_conj:}\langle G, H \rangle$. More formally the following axioms hold for all $oc \in \text{LETTER}$ and $F, G \in \text{PROP}$

$$\text{Decompose_atom} \cdot \text{Compose_atom:oc} = oc$$

$$\text{Decompose_neg} \cdot \text{Compose_neg:F} = F$$

$$\text{Decompose_conj} \cdot \text{Compose_conj:}\langle F, G \rangle = \langle F, G \rangle$$

$$\text{Decompose_disj} \cdot \text{Compose_disj:}\langle F, G \rangle = \langle F, G \rangle$$

$$\text{Atom} \cdot \text{Compose_atom:oc} = \text{TRUE}$$

$$\text{Neg} \cdot \text{Compose_neg:F} = \text{TRUE}$$

$$\text{Conj} \cdot \text{Compose_conj:}\langle F, G \rangle = \text{TRUE}$$

$$\text{Disj} \cdot \text{Compose_disj:}\langle F, G \rangle = \text{TRUE}$$

The input for our proposition evaluator also includes an interpretation $I \in \text{INTERPRETATION}$ which associates boolean values with each letter. We use the operator $\text{Assoc:LETTER} \times \text{INTERPRETATION} \rightarrow \text{IB}$ to determine the value of a given letter under an interpretation.

The output domain for our proposition evaluator is IB, which has the composition algebra

$$\langle \{\text{IB}\}, \{\text{Id, Not, And, Or}\} \rangle,$$

where

$$\text{Id:IB} \rightarrow \text{IB} \quad (\text{the identity operator}),$$

$$\text{Not:IB} \rightarrow \text{IB} \quad (\text{the usual negation operator}),$$

$$\text{And:IB} \times \text{IB} \rightarrow \text{IB} \quad (\text{the usual logical and operator}),$$

$$\text{Or:IB} \times \text{IB} \rightarrow \text{IB} \quad (\text{the usual logical or operator}).$$

A divide and conquer algorithm, called Prop_eval, for evaluating a proposition is listed in Figure 6. Here is an example computation of Prop_eval: Let F denote the representation of the proposition $(A \wedge B) \vee \neg A$ and F_1 and F_2 the

Prop_eval:<F,I>=

if

Atom:F → Id·Assoc·(Decompose_atom X Id):<F,I> []

Neg:F → Not·Prop_eval·(Decompose_neg X Id):<F,I> []

Conj:F → And·(Prop_eval X Prop_eval)·Trans·(Decompose_conj X Id2):<F,I> []

Disj:F → Or·(Prop_eval X Prop_eval)·Trans·(Decompose_disj X Id2):<F,I> []

fi

Figure 6. A Proposition Evaluator

propositions $A \wedge B$ and $\neg A$ respectively thus $F = \text{Compose_Disj}:<F_1, F_2>$. Let I be an interpretation under which letters A and B have the values TRUE and FALSE respectively.

Prop_eval:<F,I> = Or·(Prop_eval X Prop_eval)·Trans·(Decompose_disj X Id2):<F,I>
(since Disj:F holds)

= Or·(Prop_eval X Prop_eval)·Trans:<<F₁,F₂>,<I,D>>

= Or·(Prop_eval X Prop_eval):<<F₁,I>,<F₂,I>>

= Or:<FALSE,FALSE>

= FALSE

where Prop_eval:<F₁,I> and Prop_eval:<F₂,I> both evaluate to FALSE in a similar manner.

6. Concluding Remarks

We have presented a class of program schemes which provide a normal-form for expressing the structure of divide and conquer algorithms. Based on these schemes we have given a theorem relating the correctness of a divide and conquer algorithm to the correctness of its parts. The theorem gives rise to several strategies for designing divide and conquer algorithms and we used these strategies to derive several algorithms.

By using syntactic program schemes to express the structure of a diverse class of algorithms we have the disadvantage that some instances will not be in their most desirable form. However this approach to representing programming

knowledge has a number of important advantages. 1) Schemes express the essential structure of algorithms in the class in a clear and precise way. 2) Generic proofs of correctness, as provided here by Theorem 1, can be given. The correctness of a divide and conquer algorithm is reduced to the simpler task of establishing the conditions of Theorem 1. 3) By providing the essential structure of algorithms in a class schemes may suggest uniform approaches to designing them.

The design strategies we have presented involve choices which may be weakly motivated and we may need to try several alternatives before we find one which works. The resulting design process can be represented by a tree of derivation paths, some of which lead to useful algorithms, some of which are dead ends. Aside from this control problem the design strategies can be formalized for use in automatic program synthesizers. However at present it is not clear whether an adequate collection of heuristics can be found to guide an automated design process through the design space without human insight.

The top-down style of programming suggested by our design strategies can be summarized as follows. First we require a clear understanding of the problem to be solved, expressed formally by specifications. If a divide and conquer solution seems both possible and desirable we begin to explore the input and/or output domains looking for simple decomposition and composition algebras respectively. Depending on our choice we follow one of the design strategies discussed above. Using our intuition and/or proceeding formally using the separability condition we derive specifications for the unknown operators in our program. These specifications are then satisfied either by target language operators or by (recursively) designing algorithms for them. Once a correct but possibly over-structured or inefficient algorithm has been constructed we subject it to equivalence-preserving transformations resulting in a more satisfactory design.

REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1974). The Design and Analysis of Computer Algorithms, Addison-Wesley Pub. Co., Reading MA, 1974.
2. Backus, J. (1978), Can Programming be Liberated from the von Neumann Style? A Functional Style of Programming and its Algebra of Programs. CACM 21, 8(1978), pp 613-641.
3. Bentley, J. (1980), Multidimensional Divide and Conquer, CACM 23, 4(1980), 214-229.
4. Dershowitz, N., and Manna, Z., (1975), On Automating Structured Programming, Proc. Colloques IRIA on Proving and Improving Programs, Arc-et-Senans, France, July 1975.
5. Dershowitz, N. (1981), The Evolution of Programs: Program Abstraction and Instantiation, Report No. UIUCDCS-R-81-1011, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, IL, 1981.
6. Dijkstra, E.W. A Discipline of Programming. Prentice-Hall, Englewood Cliffs, NJ, 1976.
7. Gerhart, S. (1975), Knowledge about Programs: A Model and Case Study, International Conference on Reliable Software, Los Angeles, California, April, 1975, 88-94.
8. Gerhart, S. and Yelowitz, L., (1976), Control Structure Abstractions of the Backtrack Programming Technique, IEEE Trans. Software Engineering SE-2, 4(1976), 285-292.
9. Goguen, J.A., Thatcher, J.W., and Wagner, E.G., (1977), Initial Algebra Semantics and Continuous Algebras, JACM 1(24), 1977, 68-95.
10. Goguen, J.A., Thatcher, J.W., and Wagner, E.G. (1978), An initial algebra approach to the specification, correctness, and implementation of abstract data types. in Current Trends in Programming Methodology, Vol. 4. R.T. Yeh, Ed., Prentice-Hall Inc., Englewood Cliffs, NJ, 1978, 80-140.
11. Horowitz, E. and Sahni, S., (1978), Fundamentals of Computer Algorithms, Computer Science Press, Potomac, MD, 1978.
12. Smith, D.R. (1982a), Derived Preconditions and Their Use in Program Synthesis, Sixth Conference on Automated Deduction, Ed. D.W. Loveland, Lecture

Notes in Computer Science 138, Springer-Verlag, New York, 1982, pp 172-193.

13. Smith, D.R. (1982b), Top-Down Synthesis of Simple Divide and Conquer Algorithms, Technical Report NPS 52-82-011, Naval Postgraduate School, Monterey, CA, November 1982, 100 pages.

14. Smith, D.R. (1983), A Problem Reduction Approach to Program Synthesis, Submitted for publication, January 1983.

15. Yelowitz, L. and Duncan, A.G., (1977), Abstractions, Instantiations and Proofs of Marking Algorithms, Proc. ACM Symp. on Artificial Intelligence and Programming Languages, Rochester, NY, 1977, 13-21.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Douglas R. Smith, Code 52Sc Department of Computer Science Naval Postgraduate School Monterey, CA 93940	16
Chief of Naval Research Arlington, Va 22217	1

END

FILMED

4-83

DTIC